

UNIT 4. DYNAMIC PROGRAMMING

1. Introduction.

Dynamic programming techniques have in common with the voracious algorithms that are typically used to solve optimization problems and allow solving problems through a sequence of decisions. However, unlike the voracious scheme, several sequences of decisions are produced and only at the end it is known which is the best of them.

On the other hand, with respect to the divide and conquer method, in that case, the complete case is immediately attacked, which is then divided into smaller subcases. However, dynamic programming begins with the smallest, and therefore the simplest, subcases. Combining their solutions, we obtain the answers for subcases of increasing sizes, until finally we arrive at the solution of the original case.

2. Principle of optimality.

Dynamic programming techniques are based on Bellman's Principle of Optimality, which states that "any subsequence of decisions of an optimal decision sequence that solves a problem must also be optimal with respect to the subproblem it solves." This principle is not applicable to all the problems that we may find. For instance, when we seek the optimal use of limited resources. Here the optimal solution of a case may not be obtained by the combination of optimal solutions of 2 or more subcases, if the resources used in those subcases amount to more than the total available resources.

Let us suppose that a problem is solved after taking a sequence d_1, d_2, \dots, d_n of decisions. If there are d possible options for each of the decisions, a brute force technique would explore the total of possible decision sequences (combinatorial exploration). The dynamic programming technique avoids exploring all the possible sequences by means of the resolution of increasing size subproblems and storage in a table of the optimal solutions of those subproblems to facilitate the solution of the biggest problems.

For example, if the shortest path between A and C passes through B, then the part of the path from A to B must also follow the shortest path possible, as well as the part of the path that connects B with C. In this situation, the Principle of Optimality is applicable. However, if the fastest way to go from A to C takes us through B, it does not necessarily follow that it is best to go as quickly as possible from A to B, and then go as quickly as possible from B until C. If we use too much gasoline in the first half of the trip, we will have to refuel somewhere in the second part, and so we will lose more time than we have won by driving very fast. The subtrips from A to B and from B to C are not independent, because they share a resource, so the selection of an optimal solution for a subtrip may prevent us from using an optimal solution for the other. In this situation, the Principle of Optimality is not applicable.

3. The problem of the backpack.

We are given a certain number of objects and a backpack. This time we assume that objects cannot be fragmented into smaller pieces, so we can decide if we take an object or leave it, but without breaking it. For $i = 1, 2 \dots n$, suppose that object i has a positive weight w_i and a positive value v_i . The backpack can carry a weight that does not exceed

W. Our objective is to fill the backpack in such a way that the value of the included objects is maximized, respecting the capacity limitation. Let $x_i = 0$ if we decide not to take the object i or 1 if we include it. The statement of the problem would be:

Maximize $\sum_{i=1}^n x_i v_i$ with the restriction $\sum_{i=1}^n x_i w_i \leq W$, where $v_i > 0$, $w_i > 0$ and $x_i \in \{0,1\}$ for $1 \leq i \leq n$.

The voracious algorithm could not be applied because x_i has to be 0 or 1. Suppose that three objects are available, the first of which weighs 6 units and has a value of 8, while the other two weigh 5 units each and have a value of 5 each. If the backpack can carry 10 units, then the optimal load includes the two lightest objects, with a total value of 10. The voracious algorithm, would begin by selecting the object that weighs 6 units, since it is the one with the highest value per unit of weight. However, if the objects cannot be broken, the algorithm will not be able to use the remaining capacity of the backpack. The load that it produces, consists of a single object, and has a value of 8.

To solve the problem by dynamic programming, we prepare a table $V[1..n, 0..W]$ that has a row for each available object, and a column for each weight from 0 to W . In the table, $V[i, j]$ will be the maximum value of the objects that we can transport if the weight limit is j , with $0 \leq j \leq W$, and if we only include the objects numbered from 1 to i , with $1 \leq i \leq n$. The solution to this case can be found in $V[n, W]$. Once again, the Principle of Optimality can be applied. We can fill the table row by row or column by column. In the general situation, $V[i, j]$ is the maximum of $V[i-1, j]$ and $V[i-1, j-w_i] + v_i$. The first of these options corresponds to not adding the object i to the load. The second corresponds to selecting the object i , which increases the value of the load in v_i , and reduces the available capacity in w_i . Therefore, we will fill the entries in the table using the general rule:

$$V[i, j] = \max(V[i-1, j], V[i-1, j-w_i] + v_i)$$

For entries outside defined limits $V[0, j] = 0$ when $j \geq 0$, and $V[i, j] = -\infty$ for all i when $j < 0$.

Weight limit	0	1	2	3	4	5	6	7	8	9	10	11
$w_1=1, v_1=1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2=2, v_2=6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3=5, v_3=18$	0	1	6	7	7	18	19	24	25	25	25	25
$w_4=6, v_4=22$	0	1	6	7	7	18	22	24	28	29	29	40
$w_5=7, v_5=28$	0	1	6	7	7	18	22	28	29	34	35	40

The values per unit of weight are 1, 3, 3.6, 3.67 and 4. If we can only transport a maximum of 11 units of weight, then the table shows that we can compose a load whose value is 40. Table of V allows us to recover the value of the optimal load and its composition. In the example we begin by examining $V[5, 11]$. Since $V[5, 11] = V[4, 11]$ but $V[5, 11] \neq V[4, 11 - w_5] + v_5$, an optimum load cannot include object 5. Next, $V[4, 11] \neq V[3, 11]$ but $V[4, 11] = V[3, 11 - w_4] + v_4$, so an optimal charge must include object 4. Now $V[3, 5] \neq V[2, 5]$ but $V[3, 5] = V[2, 5 - w_3] + v_3$, so an optimal charge must include object 3. Continuing in this way, we find that $V[2, 0] = V[1, 0]$ and that $V[1, 0] = V[0, 0]$, so an optimal load does not include either object 2 or object 1. Thus, there is only one optimal load consisting of objects 3 and 4.

In this example, the voracious algorithm would first consider object 5, since it is the largest per unit of weight. Then the 4, but this object cannot be included without violating the capacity restriction. Then he would examine objects 3, 2 and 1 and end up with a load that would consist of objects 5, 2 and 1 with a total value of 35. Then the voracious algorithm does not work.

4. Calculation of the binomial coefficient.

Let's assume $0 \leq k \leq n$:

$$\binom{n}{k} = \begin{cases} 1 & k = 0 \text{ o } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & 0 < k < n \\ 0 & \text{en otro caso} \end{cases}$$

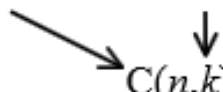
If it is directly calculated

```
function C(n,k)
  if k=0 or k=n then return 1
  else return C(n-1,k-1)+C(n-1,k)
```

Many of the values $C(i,j)$, with $i < n$ and $j < k$ are calculated again and again. Since the final result is obtained by adding a certain number of ones, the time of this algorithm is disproportionate. If we use a table of intermediate results (Pascal's triangle) then we obtain a more efficient algorithm. The table should be filled line by line. In fact, it is not even necessary to fill the entire table: just keep a vector k , which represents the current

line and update the vector from right to left. Therefore, to calculate $\binom{n}{k}$, the algorithm

requires a time that is in $O(nk)$, and a space that is in $O(k)$, if we assume that the sum is an elementary operation.

	0	1	2	3	...	$k-1$	k
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
...		
...	
$n-1$						$C(n-1, k-1) +$	$C(n-1, k)$
n							

The problem is this: in a competition there are two teams A and B that play a maximum of $2n-1$ games, and the winner is the first team to win n . There are no draws, and for any given game there is a constant probability that team A is the winner, and $q=1-p$ that team B loses and wins. Let $P(i,j)$ be the probability that the team A wins, when it still needs i more victories to get it, while Team B needs j victories to win. Before the first game, the probability of team A winning is $P(n,n)$: both teams need n victories to win. If team A wins $P(0,i)=1$, with $1 \leq i \leq n$. If team B wins $P(i,0)=0$ with $1 \leq i \leq n$. So:

$$P(i, j) = pP(i-1, j) + qP(i, j-1) \quad i \geq 1, j \geq 1$$

P(i,j) can be obtained in the following way:

```
function P(i,j)
    if i=0 then return 1
    else if j=0 then return 0
    else return pP(i-1,j)+qP(i,j-1)
```

Let $T(k)$ be the time needed in the worst case to calculate $P(i,j)$, with $k=i+j$. We see that:

$$T(1) = \mathbf{c}$$

$$T(k) \leq 2T(k-1) + d, \quad k > 1$$

with c and d constants. Rewriting $T(k-1)$ in terms of $T(k-2)$ and so on, we get:

$$T(k) \leq 4T(k-2) + 2d + d, \quad k > 1$$

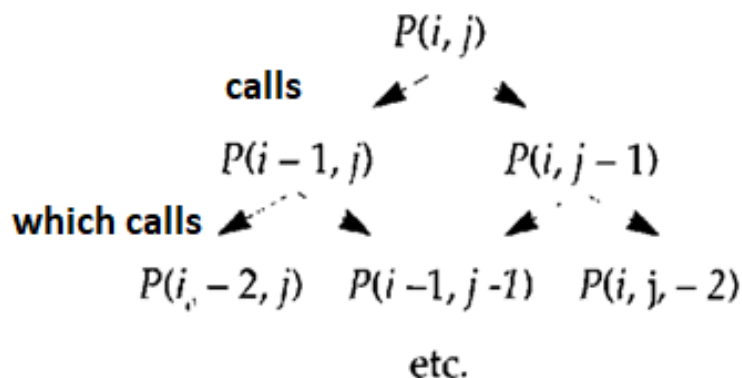
...

$$\leq 2^{k-1}T(1) + (2^{k-2} + 2^{k-3} + \dots + 2 + 1)d =$$

$$= 2^{k-1}c + (2^{k-1} - 1)d =$$

$$= 2^k (c/2 + d/2) - d$$

$T(k)$ is in $O(2^k)$ which is $O(4^n)$ if $i=j=n$.



In this way, the value of each $P(i,j)$ is calculated many times. It is better to use a table to store those values. To accelerate the algorithm, we declare a matrix of the appropriate size and then fill in the entries, but instead of line by line we work diagonal to diagonal. The algorithm has to fill an $n \times n$ matrix, then the time is $O(n^2)$. Just like the Pascal triangle we could implement the algorithm so that the time is $O(n)$. The algorithm to calculate $P(n,n)$ would be:

```
function series(n,p)
    matrix P[0..n,0..n]
    q=1-p
    {We fill from the left corner to the main diagonal}
    for s=1 to n do
        P[0,s]=1
        P[s,0]=0
        for k=1 to s-1 do
            P[k,s-k]=pP[k-1,s-k]+qP[k,s-k-1]
        {We fill from below the main diagonal to the right corner}
    for s=1 to n do
        for k=0 to n-s do
            P[k,s-k]=pP[k-1,s-k]+qP[k,s-k-1]
    return P(n,n)
```

6. Changing money.

The voracious algorithm is very efficient but works only in a limited number of cases. With certain monetary systems or when there is a lack of coins of a certain denomination or their number is limited, the algorithm can find a response that is not optimal or does not find an answer. Example: we have coins of 1, 4 and 6 units. If we have to change 8 units, the voracious algorithm returns a coin of 6 units and two of a unit, that is, three coins. But we can do better by giving two four-unit coins. Suppose we have an unlimited number of coins of n different denominations and that a currency of denomination i , with $1 \leq i \leq n$ has a value of d_i units, $d_i > 0$. We have to give the client currencies worth N units using as few coins as possible.

The problem can be solved by dynamic programming. We prepare a table $c[1..n,0..N]$ with a row for each possible denomination and a column for quantities ranging from 0 units to N units. $C[i,j]$ is the minimum number of coins needed to pay a quantity of j units, with $1 \leq j \leq N$, using only denomination coins from 1 to i , with $1 \leq i \leq n$. Solution: $c[n,N]$ if all we need to know is the number of coins needed. We start with $C[i,0]=0$ for all values of i . The table of the example would be the following:

Quantity	0	1	2	3	4	5	6	7	8
$d_1=1$	0	1	2	3	4	5	6	7	8
$d_2=4$	0	1	2	3	1	2	3	4	2
$d_3=6$	0	1	2	3	1	2	1	2	2

It gives us the solution for all cases involving a payment of 8 units or less. And the algorithm would be the following:

```

function coins(N)
    {Returns the minimum amount of coins necessary to change N units.
    Vector d[1..n] especifices the kind of coins: in the example there are coins
    of 1, 4 and 6 units}
    vector d[1..n]=[1, 4, 6]
    matrix c[1..n, 0..N]
    for i=1 to n do c[i,0]=0
    for i=1 to n do
        for j=1 to N do
            if i=1 and j<d[i] then c[i,j]=+∞
            else if i=1 then c[1,j]=1+c[1,j-d[1]]
            else if j<d[i] then c[i,j]=c[i-1,j]
            else c[i,j]=min(c[i-1,j],1+c[i,j-d[i]])
    return c[n,N]

```

7. Minimum paths.

Let $G=(N, E)$ be a directed graph, N is the set of nodes and E the set of edges. We want to calculate the shortest path length between each pair of nodes. Suppose that the nodes of G are numbered from 1 to n , so $N=\{1,2, \dots, n\}$, and let L be the matrix that gives the edge lengths, with $L[i,i]=0$ for $i=1,2, \dots, n$, $L[i, j] \geq 0$ for all i and j , and $L[i,j]=\infty$ if the edge (i,j) does not exist. The Principle of Optimality can be applied: if k is a node of the minimum path between i and j , then the part of the path that goes from i to k , and the part of the path that goes from k to j must be optimal too. We build a matrix D that gives the shortest path length between a pair of nodes. The algorithm gives D the initial value L , that is, the direct distances between nodes, and then performs n iterations. After iteration k , D gives the length of the shortest paths using only the nodes $\{1,2, \dots k\}$ as intermediate nodes. After n iterations, D gives us the length of the shortest paths that use some of the nodes of N as an intermediate node, which is the desired result. In iteration k , the algorithm must check for each pair of nodes (i,j) whether or not there is a path that goes from i to j passing through node k , and that is better than the current optimal path that passes only through the nodes $\{1,2,\dots,k-1\}$. If D_k represents the matrix D after the k -th iteration ($D_0=L$), then the necessary verification must be implemented in the form:

$$D_k[i, j] = \min(D_{k-1}[i, j], D_{k-1}[i, k] + D_{k-1}[k, j])$$

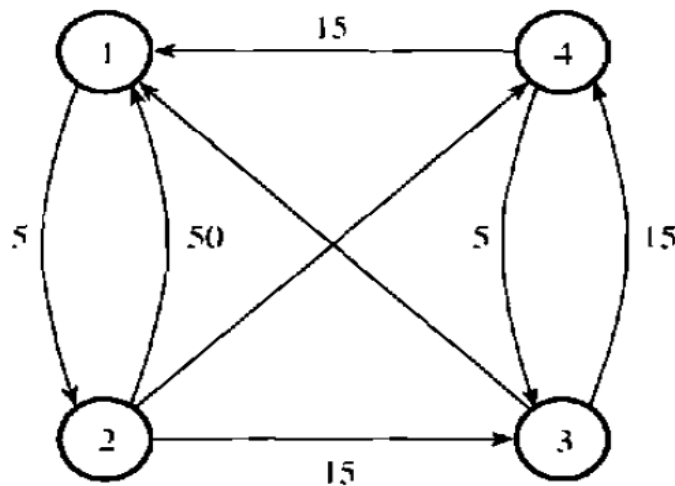
It has been used the fact that an optimal path that passes through k will not visit k twice. This algorithm is known as the Floyd algorithm and is as follows:

```

function Floyd(L[1..n, 1..n]): matrix[1..n, 1..n]
    matrix D[1..n, 1..n]
    D=L
    for k=1 to n do
        for i=1 to n do
            for j=1 to n do
                D[i,j]=min(D[i,j],D[i,k]+D[k,j])
    return D

```

Example:



$$D_0 = L = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \infty & 0 & 15 \\ 15 & \infty & 5 & 0 \end{pmatrix}$$

$$D_1 = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

$$D_2 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

$$D_3 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 45 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

$$D_4 = \begin{pmatrix} 0 & 5 & 15 & 10 \\ 20 & 0 & 10 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

In the k -th iteration, the values of the k -th row and the k -th column of D do not change, because $D\{k,k\}$ is always 0. Therefore, it is not necessary to protect these values when updating D . This allows us to use a single D of size $n \times n$.

The complexity of the algorithm is $O(n^3)$. With the Dijkstra algorithm the time is the same but the simplicity of Floyd's algorithm makes it have a smaller hidden constant, and therefore be faster in practice. Normally, we want to know which is the shortest path, and not just its length. In this case, we will use a second matrix P , whose elements all have an initial value of 0. The most internal loop of the algorithm becomes

$$\begin{aligned} &\text{if } D[i,k] + D[k,j] < D[i,j] \text{ then} \\ &\quad D[i,j] = D[i,k] + D[k,j] \\ &\quad P[i,j] = k \end{aligned}$$

When the algorithm is stopped, $P[i,j]$ contains the number of the last iteration that resulted in a change in $D[i,j]$. To recover the shortest path from i to j , examine $P[i,j]$. If $P[i,j]=0$, then $D[i,j]$ has never changed, and the minimum path passes directly along the edge (i,j) ; otherwise, if $P[i,j]=k$, then the shortest path from i to j passes through k . We recursively examine $P[i,k]$ and $P[k,j]$ to find any other possible intermediate node that is on the shortest path. In the previous example P happens to be

$$P = \begin{pmatrix} 0 & 0 & 4 & 2 \\ 4 & 0 & 4 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}.$$

Given that $P[1,3]=4$, the minimum path from 1 to 3 passes through 4. Examining $P[1,4]$ and $P[4,3]$, we discover that between 1 and 4 we must pass through 2, but that from 4 to 3 we passed directly. The routes from 1 to 2 and from 2 to 4 are direct. Therefore, the minimum path from 1 to 3 is 1, 2, 4, 3.